Factoring Small to Medium Size Integers: An Experimental Comparison

Jérôme Milan

Factoring Small to Medium Size Integers: An Experimental Comparison

Jérôme Milan

INRIA, École Polytechnique, CNRS
jerome.milan@lix.polytechnique.fr

Abstract. We report on our experiments in factoring integers from 50 to 200 bit with the NFS postsieving stage or class group structure computations as potential applications. We implemented, with careful parameter selections, several general-purpose factoring algorithms suited for these smaller numbers, from Shanks's square form factorization method to the self-initializing quadratic sieve, and revisited the continued fraction algorithm in light of recent advances in smoothness detection batch methods. We provide detailed timings for our implementations to better assess their relative range of practical use on current commodity hardware.

1 Introduction

It is almost cliché to recall that integer factorization is a challenging algorithmic problem, be it from a purely theoretical or from a more pragmatic standpoint. On the utilitarian side, the main interest in integer factorization certainly stems from cryptography given the ubiquity of the RSA cryptosystem [4], based on the premise that factoring large integers is computationally impracticable. Moreover, factorization is one of the main ingredients in a lot of number theoretical algorithms – for instance, computing the maximal order of a number field requires to factor the discriminants.

In this paper, we focus on factorizations of small to medium sized composites such as those arising in most sieving methods, be it in the context of record factorizations with the Number Field Sieve [8], discrete logarithms [16], or class group structure computations [6,11,3]. These numbers are consequently known to have no small factors and selected to be the product of very few primes (typically 2), usually of roughly comparable sizes. We reviewed general-purpose factoring methods most suitable to this task and implemented and fine-tuned Shanks's SQUFOF algorithm, McKee's enhanced Fermat described in [13], ECM, and the subexponential CFRAC and SIQS improved with Bernstein's batch method [2] to identify the smooth residues.

The next section briefly describes the implemented algorithms. We then give a detailed account of our timing measurements and discuss the implication of these results from a down-to-earth perspective.

2 Implemented algorithms

In this section, we identify and briefly describe the main phases of the implemented algorithms, with the aim of understanding the results presented in the third section. For a more detailled theoretical background, see the referenced literature.

2.1 Shanks's SQuare Form Factorization (SQUFOF)

SQUFOF [9], proposed in the mid-seventies by Shanks as an alternative to the continued fraction method, factors a number N in $O(N^{1/4})$. A nice feature is that most of the computations involve numbers less than $2\sqrt{N}$ which makes it particularly suited to factor (at most) double precision numbers.

SQUFOF is based on the theory of quadratic forms $F(x,y) = ax^2 + bxy + cy^2 \equiv (a,b,c)$, more precisely, on the underlying structure of cycles of reduced forms $(a_i, b_i, c_i) = \rho^i(a_0, b_0, c_0)$ where ρ is the standard reduction operator. Finding a point of symmetry in such a cycle leads to a simple relation potentially giving a factor of N. The theory of quadratic forms is tightly linked to continued fractions and the reduction operator can be expressed in a similar formalism. Given a number to factor N, we recall the following relations arising from the development of $\sqrt{N}$ in continued fractions:

$$q_0 = \lfloor N \rfloor \quad , \quad q_i = \left\lfloor \frac{q_0 + P_i}{Q_i} \right\rfloor \text{ for } i > 0 \tag{1}$$

$$P_0 = 0 \quad , \quad P_1 = q_0 \tag{2}$$

$$P_i = q_{i-1}Q_{i-1} - P_{i-1} \text{ for } i > 1 \tag{3}$$

$$Q_0 = 1 \quad , \quad Q_1 = N - q_0^2 \tag{4}$$

$$Q_i = Q_{i-2} - q_{i-1}(P_{i-1} - P_i) \text{ for } i > 1 \tag{5}$$

Moreover we have the pivotal equality:

$$N = P_m^2 + Q_{m-1}Q_m \tag{6}$$

The principal cycle of reduced forms is given by the set of forms $\rho^i(F_0) = ((-1)^{(i-1)}Q_{i-1},\ 2P_i,\ (-1)^iQ_i)$ with the principal form $F_0 = (1, 2q_0, q_0^2 - N)$. We will now recall the main steps of the algorithm.

[1 - forward: find square form]
Move forward through the cycle starting with the principal form F_0 until we identify a square form $F_n = \rho^n(F_0) = (-Q, 2P, S^2)$.

[2 - invsqrt: inverse square root]
Set $F^{-1/2} = (-S, 2P, SQ)$ and compute the reduction $G_0 = (-S_{-1}, 2R_0, S_0)$ where $S_{-1} = S$, $R_0 = P + S\lfloor (q_0 - P)/S \rfloor$ and $S_0 = (N - R_0^2)/S$.

[3 - reverse: find symmetry point]
Using (1) to (5), reverse cycle through the quadratic forms $G_i = \rho^i(G_0) = ((-1)^{(i-1)}S_{i-1},\ 2R_i,\ (-1)^iS_i)$ to find a symmetry point, *e.g.* a pair of forms

G_m, $G_m + 1$ with $R_m = R_{m+1}$ (this happens for $m \approx n/2$). Using (3), (5) write $R_m = t_m S_m/2$ and since $N = R_m^2 + S_{m-1}S_m$, we obtain a factorization of N: $N = S_m \cdot (S_{m-1} + S_m t_m^2/4)$.

As in other factoring methods, one can use a multiplier k, leading to factor kN instead of N hoping to achieve some speed up. We refer the reader to [9] for a full theoretical background on SQUFOF. Finally, note that in step 3, we have a rough estimate of the number of forms to explore. Thus by using form compositions, one can jump in the cycle to get quickly in the vicinity of the point of symmetry, a strategy dubbed "fast return" by Shanks [19].

2.2 McKee's speedup of Fermat's algorithm

J. McKee proposed in [13] an improvement to Fermat's venerable algorithm. Heuristically running in $O(N^{1/4})$ (instead of $O(N^{1/2})$), it is described as a possibly faster alternative to SQUFOF.

Assuming that N has no factor less than $2N^{1/4}$ (which is perfectly acceptable for the applications we target), define $b = \left\lceil \sqrt{N} \right\rceil$ and $Q(x, y) = (x + by)^2 - Ny^2$. As in Fermat's algorithm, we seek x and y such that $Q(x, y)$ is a square so that $\gcd(x + by - \sqrt{Q(x, y)}, N)$ is potentially a proper factor of N. The "greedy" variant proceeds as follows:

[1. Compute modular square root]
Choose a prime $p > 2N^{1/4}$ and set x_0 and x_1 as the solutions to $Q(x, 1) \equiv 0 \pmod{p^2}$.

[2. Find square]
For $x_i \in [x_0, x_1]$: Set $x = x_i$ and $y = 1$. While $Q(x, y)$ is not a square, set $r = \lceil p^2/x \rceil$, $x = xr - p^2$ and $y = r$. Abort the loop when y exceeds a given threshold y_{max} in the order of $N^{1/4}$.
If no factor is found, go back to step 1 and choose a different prime p. Abort the algorithm when p reaches a chosen bound.

2.3 The Elliptic Curve Method (ECM)

ECM was proposed by H.W. Lenstra in 1985 [10] and saw numerous subsequent improvements, most notably by P. Montgomery [14] and R. Brent [5]. ECM's running time essentially depends on the size of the factor to be found, like Pollard's rho or $p - 1$ methods. Given a number N having p as its smallest factor, ECM's asymptotic complexity is given by $L_p(1/2, \sqrt{2}) \cdot M(N)$ where $L_x(\alpha, c) = \exp((c + o(1)) \cdot (\log p)^\alpha \cdot (\log \log p)^{1-\alpha})$ and $M(N)$ is the cost of multiplication modulo N.

ECM is actually very similar to the $p - 1$ algorithm but works on a group defined by an elliptic curve, which makes it possible to switch to another group (*i.e.* curve) if one try fails. We sketch here the main idea of the method using the so-called "standard continuation".

[**input**]
The number to factor N and two bounds B_1 and B_2.
[**1 - ccurve: choose an elliptic curve**]
Choose an elliptic curve E and P_0 a non-torsion point on the curve. All operations on the curve are performed modulo N.
[**2 - phase_1: first phase**]
Let $k = \Pi_{p_i<B_1}(p_i^{\lfloor \log(B_1)/\log(p_i) \rfloor})$. Compute P $= [k]P_0$. Check if $P = \mathcal{O}$ (the point at infinity). Since this involves an inversion modulo N (and hence the computation of a gcd modulo N), this will give a factor of N if $P = \mathcal{O}$.
[**3 - phase_2: second phase (standard continuation)**]
For each prime $p_i \in [B_1, B_2]$, compute $[p_i]P$. If $[p_i]P = \mathcal{O}$, we have found a factor of N. If this second phase fails, retry with another curve and / or other bounds.

The original version of ECM did not include a second phase. In that case, ECM will split N if the order of the group on the curve has all of its prime factors smaller than B_1. Using a second phase makes it possible to split N if all but one of the prime factors of the order are less than B_1 and if the larger one is less than B_2.

2.4 The Continued FRACtion algorithm (CFRAC)

Introduced in 1975 by Morrison and Brillhart in [15], CFRAC is a subexponential algorithm running in $L_N(1/2, \sqrt{2})$. It belongs to the "congruence of squares" family of algorithms, so-called because they look for relations of the form $X^2 \equiv Y^2 \pmod N$ to factor a composite N. However, they do not attempt to find directly such (X, Y) pairs, focusing instead on the easier task to find several congruences c_i of the form $x_i^2 \equiv y_i \pmod N$ at the expense of finding afterwards a sub-collection $\{c_j\}$ such that $\prod_j y_j$ is a square. Finding such a sub-collection is achieved by factoring the y_i on a factor base $\mathcal{B} = \{p_1, \ldots, p_l\}$ and solving a linear algebra system over $\mathbb{F}_2$. The smoothness condition on the y_i is usually relaxed to allow them to be the product of a $\mathcal{B}$-smooth number by one or more primes greater than p_l. These are the so-called large prime variations.

CFRAC presents the advantage to generate residues y_i less than $2\sqrt{N}$. Congruences are found by computing the partial quotients q_i of the continued fraction expansion of $\sqrt{N}$ as given in (1) to (5) and the numerator of the i-th convergent given by:

$$A_0 = 1 \quad , \quad A_1 = q_0 \quad , \quad A_i = q_i A_{i-1} + A_{i-2} \quad \text{for } i > 1 \tag{7}$$

The congruences $\{c_i\}$ are then given by the relation:

$$(-1)^i Q_i = A_i^2 \pmod N \tag{8}$$

where Q_i is given in (4), (5). The factor base $\mathcal{B}$ is obtained by keeping only the primes p_i such that N (or kN if a multiplier is used) is a quadratic residue mod p_i. The multiplier k, if any, is chosen so that the factor base contains as many small primes as possible. The main steps of CFRAC (without large primes) can be summarized as follows:

[1 - genrel: generate relations]
Expand $\sqrt{kN}$ as continued fractions to generate congruences c_i as in (8).

[2 - selrel: select relations]
If y_j is $\mathcal{B}$-smooth, keep the relation c_j and factor $y_i = \prod_{j=1}^{l} p_j^{e_{ij}}$ on the base. This gives a new row in a binary matrix M whose coefficients are given by the e_{ij} mod 2. Go back to step 1 to generate new relations until we have at least $l + \delta$ relations.

[3 - linalg: solve linear system]
Find the kernel of the matrix M. This yields δ sub-collections of relations $\{c_j\}$ for which $\prod_j y_j$ is a square.

[4 - dedfac: deduce factors]
For each sub-collection found, compute $Y_i^2 = \prod_j y_{ij}$ and $X = \prod_j x_{ij}$. A possible factor of N is given by $\gcd(X - Y, N)$.

The large prime variations only alter step 2 of the algorithm. We refer to [15] for a description of the single large prime variation which is the only one we implemented, albeit in the context of batched smooth number detection (cf 2.6).

2.5 The Self-Initializing Quadratic Sieve (SIQS)

SIQS is the last sibling of the quadratic sieve (QS) methods. It basically differs from CFRAC in the way the relations c_i are generated. Compared to CFRAC, its chief advantage lies in the introduction of a sieving process which, by discarding most of the non-smooth y_i, lowers its complexity to $L_N(1/2, 3/\sqrt{8})$.

In the QS methods, the residues y_i are taken as the values of quadratic polynomials $g_i(x)$. These polynomials are constructed so that if a prime p in the factor base divides $g_i(x_0)$ then p also divides $g_i(x_0 + jp)$, $j \in \mathbb{Z}$. This allows to use a sieve, tagging values of x for which $g(x)$ is likely to be smooth. The main steps of the vanilla QS are recalled below:

[1 - polyinit: initialize polynomial]
Take as polynomial $g(x) = (x + \lceil N \rceil)^2 - N$ and "initialize" it, *i.e.* compute $\{x_{0i}\}$ and $\{x_{1i}\}$ such that $g(x_{0i}) \equiv 0 \pmod{p_i}$ and $g(x_{1i}) \equiv 0 \pmod{p_i}$.

[2 - fill: fill sieve]
A sieve is used to tag x values for which $g(x)$ is divisible by each p_i of the base (for example, initialize the sieve with zeroes and add $\lfloor \log p_i \rceil$ at the positions $x_{0i} + jp_i$ and $x_{1i} + jp_i$, $j \in \mathbb{Z}$).

[3 - scan: scan sieve and select relations]
Values of x for which $g(x)$ is smooth have a sieve value nearly equal to $\log g(x)$. In practice a small correction term is applied to account for rounding errors, and the fact that we do not sieve with prime powers. For each x_i fulfilling this criteria, compute $g(x_i)$ and check if it is $\mathcal{B}$-smooth. Each smooth $g(x_i)$ gives a new line in an $(l + \delta) \times l$ matrix like in the CFRAC method. Go back to step 2 while the matrix is not filled.

[4 - linalg & 5 - dedfac]
These last two steps are generic to "congruence of squares" methods and are thus similar to CFRAC's.

QS's main drawback is that the generated $\{y_i\}$ are no longer bounded above by $2\sqrt{N}$ as in CFRAC, but grow linearly, which lowers the probability to find smooth residues. A variant known as MPQS (Multiple Polynomial QS) sieves with several polynomials $g_{ab}(x) = (ax + b)^2 - N$, discarding one g_{ab} when the residues get too large, which leads to sieve on a smaller, fixed interval $[-M, +M]$. However switching polynomials requires recomputing the solutions to $g_{ab} \equiv 0 \pmod{p_i}$, which can become costly.

SIQS (basically a variant of MPQS) bypasses this problem by judiciously choosing a family $\{g_{ab_i}\}$ such that a new polynomial $g_{ab_{i+1}}$ can (to a certain extent) be quickly initialized from g_{ab_i}. We voluntarily put aside the details and refer the reader to [7] for a full description of SIQS.

[1 - polyinit: initialize polynomial]

[1.1. Full polynomial initialization]
Choose a as explained in [7]. The first polynomial g_{ab_0} must be fully initialized as in MPQS.

[1.2. "Fast" polynomial initialization]
If the current polynomial is g_{ab_i} with $i < 2^s$ (with s being the number of prime factors of a), a new polynomial $g_{ab_{i+1}}$ can be quickly initialized from g_{ab_i} otherwise, goto step 1.1.

[2 - fill: fill sieve]
This is similar to the 'fill' stage in QS, with the exception that we only sieve on a fixed interval $[-M, +M]$. If this interval has already been completely sieved, go back to step 1 to switch to a different polynomial.

[3 - scan & 4 - linalg & 5 - dedfac]
These last step are performed in the same way as in the basic QS.

2.6 Identifying smooth residues

Testing the smoothness of a lot of residues is a major bottleneck of the "congruence of squares" methods. While sieving methods greatly mitigate its extent, it remains by far the most time-consuming stage of the CFRAC algorithm.

Identifying the smooth residues has traditionally been achieved by trial-division. The early abort variation can significantly reduce the cost of this phase by beginning to trial-divide with only a fraction of the primes in the base and discarding the partially factored residues if they exceed a given bound.

In [2], D. Bernstein introduced a simple but efficient batch algorithm to find the $\mathcal{B}$-smooth parts of a large set of integers $\{x_i\}$. His method runs in $O\big(b \cdot \log^2(b) \cdot \log(\log(b))\big)$ where b is the total number of bits in $\mathcal{B}$ and $\{x_i\}$. We reproduce below almost verbatim the algorithm described in [2] (which we will dub "smoothness batch") as a convenience for the reader.

[Input] A factor base $\mathcal{B} = \{p_1, \ldots, p_l\}$ and a set of integers $S = \{x_1, \ldots, x_m\}$.
[Output] The set of integers $S' = \{x'_i | x'_i$ is the $\mathcal{B}$-smooth parts of $x_i\}$.

[0 - precomp: precompute primes product]
Precompute $z = p_1 \times \cdots \times p_l$ using a product tree (this step is performed only once).

[1 - prodtree: compute the $\{x_i\}$ product]
Compute $X = x_1 \times \cdots \times x_m$ using a product tree.

[2 - remtree: product modulo the $\{x_i\}$]
Compute $Z' = \{z'_i | z'_i = z \bmod x_i\}$ using a remainder tree and the previously computed product tree.

[3 - powm: compute modular powers]
Compute $Y = \{y_i | y_i = {z'_i}^{2^e} \bmod x_i\}$ where $e > 0$ is the smallest integer such that $2^{2^e} \geq x_i$.

[4 - gcd: compute smooth parts]
$S' = \{x'_i | x'_i = \gcd(x_i, y_i)\}$. Note that if $y_i = 0$ then x_i is $\mathcal{B}$-smooth, otherwise the cofactor x_i/x'_i may be prime and hence considered as a large prime.

This algorithm does not give the factorization of the x_i on the base but only identifies smooth numbers. However for our applications, this phase (referred to as "factres" in the plots of section 3) is generally negligible and can be carried out via simple trial division without much impact on the total running time.

3 Implementation and results

Our programs were developed and benchmarked on a 64-bit AMD Opteron 250 workstation with 2GB of RAM running under the GNU/Linux operating system. They are single-threaded and aimed to be run on a single core/processor. Multi-precision computations were performed using the GMP library version 4.2 patched with Pierrick Gaudry's assembly routines for Opteron processors. The programs, written exclusively in C99 for portability reasons, were compiled with the `gcc` compiler version 4.1 using the optimization flags `'-march=opteron -O3'`.

Most composites used in our experiments are taken as products of two random primes of similar sizes. While restrictive, such a choice represents the bulk of the composites in our targeted applications.

3.1 Linear algebra

For the size of the numbers we are interested in, the linear algebra phase does not significantly impact the total running time. We thus settled for Gaussian elimination following the algorithm given in [17].

3.2 SQUFOF

Our implementation follows closely the continued fraction description given in [9]. In order to use mostly single precision computations, our program is restrained to factor numbers fitting in two machine words.

Factorization is first attempted without using a multiplier. If this first attempt fails (which happens about 5% to 10% of the time) we perform a sequential race using the 15 square free multipliers suggested in [9], which, for all

intent and purpose, is guaranteed to succeed as we observed no failure during our experiments.

To carry out the "fast return", we adapted the "large step" algorithm described by Williams and Wunderlich in [21] in the context of the parallel generation of residues for CFRAC. To our knowledge, this is the first time the "fast return" was implemented in such a way.

Fig. 1 a) shows the mean timings for SQUFOF in a range from 45 to 80 bits. 1000 composites were factored for each size shown. The 'reverse' step of our program only performed the "fast return" variation if the estimated number of forms to scan was greater than a given bound n_{FR}, to take into account the slower multi-precision computations involved and the fact that the form cycle should then be scanned in both directions. Our implementation uses $n_{FR} = 4096$ which has been determined experimentally. It should be noted that the improvement obtained by using a "fast return" is almost negligible for composites under 60 bit.

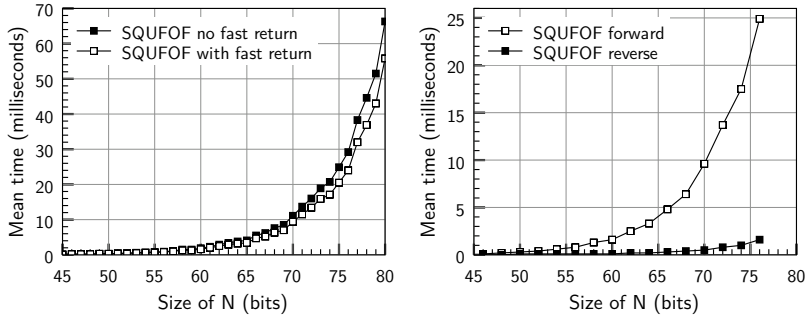

Fig. 1: Left: SQUFOF timings from 45 to 80 bits (1000 composites per size) with and without fast return. Right: Relative impact of the 'forward' and 'reverse' steps for SQUFOF with fast return.

Our SQUFOF implementation was then compared to the PARI[1] computer algebra system (SVN version early december 2009) and YAFU[2] (version 1.10), an optimized fork of Msieve[3]. Fig. 2 shows very similar performances. While our version is the only one including a "fast-return" variant, its benefit is only visible for larger composites which cannot be factored with PARI's and YAFU's versions as they are limited to single precision composites.

3.3 McKee's Fermat variation

We implemented the "greedy" variant of McKee's algorithm following the same set-up as described in [13]. Our program uses single precision functions whenever

[1] http://pari.math.u-bordeaux.fr/
[2] http://sites.google.com/site/bbuhrow/home
[3] http://sourceforge.net/projects/msieve/

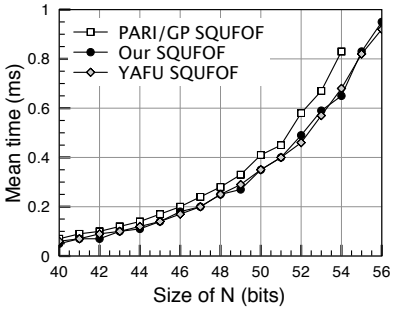

Fig. 2: Comparison of SQUFOF with PARI's and YAFU's implementations.

possible and is restricted to double precision integers at most. In Fig. 3, we provide timings for a race with the multiplier lists $\{1, 3\}$ and $\{1, 3, 5, 7, 11\}$ for tiny 45 to 60 bit composites. We did not observe any failure in that size range.

Our measurements are in disagreement with McKee's in that SQUFOF appears to be systematically faster. Two reasons could account for such a discrepancy. First, timings given in [13] are restricted to a tiny sample of composites which may not be statistically significant. Second, [13] uses Maple's SQUFOF as a reference, but this implementation is actually very poor (several times slower than ours).

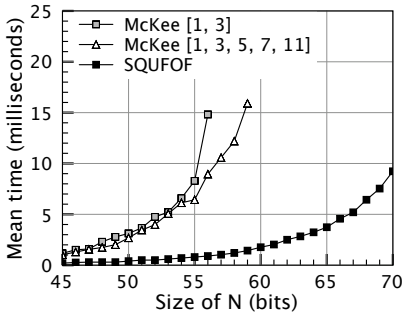

Fig. 3: McKee's "greedy" Fermat timings using the multipliers $\{1, 3\}$ and $\{1, 3, 5, 7, 11\}$ averaged over 100 composites per size, compared to SQUFOF's.

3.4 ECM

We implemented ECM using the commonly used Montgomery's point representation and chose curves given by Suyama's parameterization [14]. The implementation of the first phase uses the Montgomery ladder algorithm. Our second phase follows the so-called "improved" standard continuation to speed up point multiplication by storing precomputed points in a similar fashion as in [12].

The bounds B_1 and B_2 were adjusted specifically to composites with factors of comparable size. Fig. 4 shows timings obtained for ECM together with a comparison with SQUFOF. It should be mentioned that our ECM implementation is generic, and thus multi-precision, while most of SQUFOF is single-precision which is definitely an advantage when factoring tiny composites.

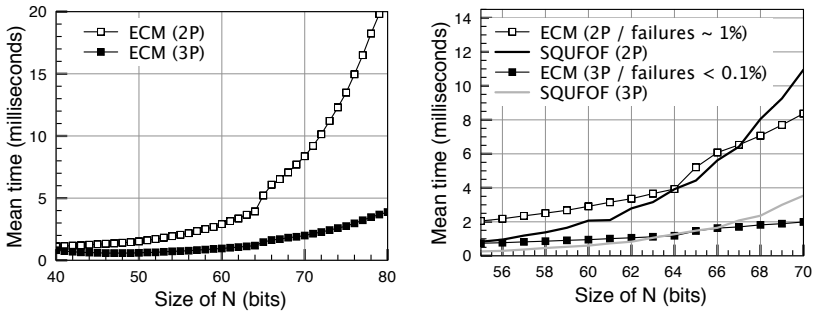

Fig. 4: Left: ECM timings averaged over 100 composites and 100 executions per size for composites with 2 prime factors (2P) and 3 prime factors (3P) of similar sizes and a failure rate from 0.5 to 2%. Right: Comparison with SQUFOF.

3.5 CFRAC

Our implementation of CFRAC is adapted from the description given in [15] and uses the single large prime variation.

We chose a multiplier $k < 100$ to have $kN \equiv 1 \pmod 8$ and to maximize the number of small primes for which kN is a quadratic residue using Silverman's modified Knuth-Schroeppel function [20].

Selection of the smooth residues is achieved either via trial-division (with or without early abort) or using the aforementioned "smoothness batch" algorithm followed by a trial-division step. Batches are performed with 128 residues at a time.

Early abort is programmed as suggested by Pomerance in [18]. If p_k is the greatest prime in the base, we first begin to trial-divide with primes less than $\sqrt{p_k}$ and discard any partially factored residues over a given bound. Pomerance's asymptotic analysis gives N^{1-c}, with $c = 1/7$, as the optimal bound. However we found this theoretical value to be of little practical use. After experimenting with some multiples of c's "optimal" value, we settled for $c = 4/7$, which works best for us.

Our timings obtained for CFRAC are presented in Fig. 5. The sizes of bases given in Table 1 were determined for each strategy independently. As expected, using batches allows to use much larger bases, hence the gain in performance.

Fig. 6 a) shows the relative impact of the different steps involved in CFRAC. Despite the use of smoothness batches, the relation selection still dominates the running time by a fair margin. Note that the actual factorization of the residues

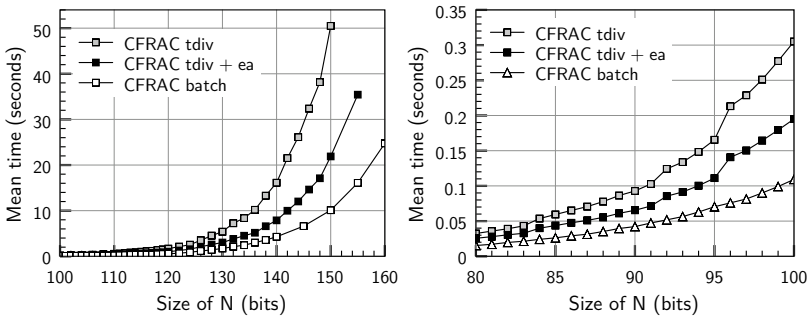

Fig. 5: Left: CFRAC timings obtained with smooth selection of the residues by trial division only (tdiv), by trial division and early abort (tdiv+ea) and by smoothness batch (batch). Right: Zoom on the 80-100 bits range.

CFRAC tdiv (+ ea)

Size of N (bits)	Size factor base
80	128
95	160
105	192
110	256
130	384
140	448
145	512

CFRAC batch

Size of N (bits)	Size factor bases
60	128
92	256
110	512
125	1024
145	2048
172	4096

Table 1: Left: Sizes of the factor bases used for CFRAC with trial division (with or without early abort). Right: Sizes of the bases used for CFRAC with batches.

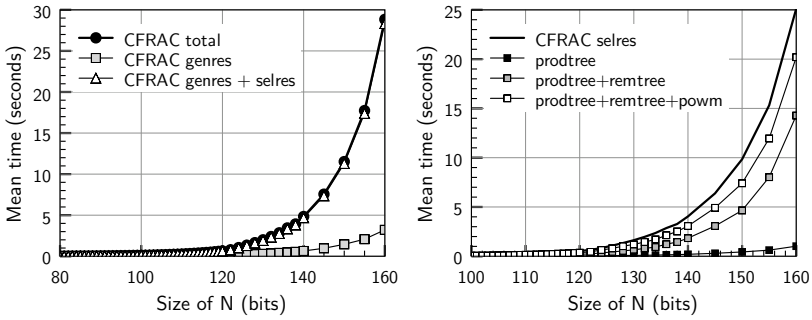

Fig. 6: Left: Timings for the first two steps of the CFRAC algorithm with batches (generate relations and select relations) compared to the total running times. Right: Timings for the different steps of the smoothness batches.

is negligible just like the linear algebra phase even though Gaussian elimination is used. Fig. 6 b) shows the contribution by each step of the smoothness batches. The real bottleneck comes from the computations of the remainder trees and, to a much lesser extent, the modular powers at each node.

3.6 SIQS

Our main references for SIQS are [20,7]. Factor bases are chosen as in CFRAC.

SIQS polynomial selection follows the Carrier-Wagstaff method [1]. Special care was taken to tune it properly to give good results even for tiny numbers (*e.g.* 60 bits or so). Most of the leading coefficients obtained are within a few percent of the ideal target values.

Note that we took the unusual approach to use batches to select relations for SIQS, even though the use of a sieve theoretically makes it redundant. The basic idea is to relax the sieving criteria to collect more residues hoping a) to find more relations via the large prime variation and b) to cut the time needed to fill the sieve.

Conceptually, residues are computed from the values of the sieve S such that $S[x] \geq \tau$. Optimal values of τ where determined experimentally instead of using the usual approximation $\log(M\sqrt{M})$.

Fig. 7 shows the running time of SIQS for the empirical selection of parameters given in Table 2. Details of the relative timings for each steps of the algorithm are given in Fig. 8. Sieve filling and polynomial initialization account for the major part of the relation collection, typically between 50% and 80% of the relation collection ("collect") time.

SIQS implementations are compared in Fig. 9. Our version is quite competitive even if YAFU is about 20% faster for 200 bits composites. That said, our SIQS nicely scales down to the smaller numbers while our concurrents revert to the standard QS or MPQS variant.

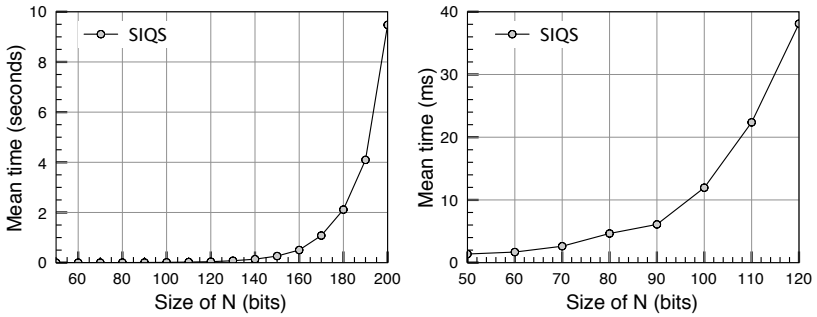

Fig. 7: SIQS timings over the 50-200 bit range. Each data point is obtained from an average over 20 to 100 composites.

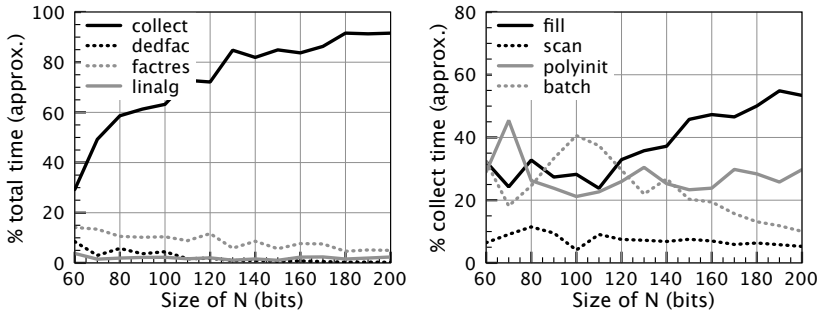

Fig. 8: Timings for each step of SIQS. The left figure shows the most time-consuming stages. The right figure breaks down the contributions to the relation collection phase.

SIQS			
Size of N (bits)	Size of base	M	τ
60	60	10000	30
80	110	14000	31
100	200	30000	36
120	400	30000	43
140	700	49152	48
160	1300	65536	56
180	2100	65536	62
200	4600	196608	71

Table 2: Parameters used for SIQS for a selected set of composite sizes. Note that the best τ values are much lower than in the traditional cases where trial division is used to complete the sieving.

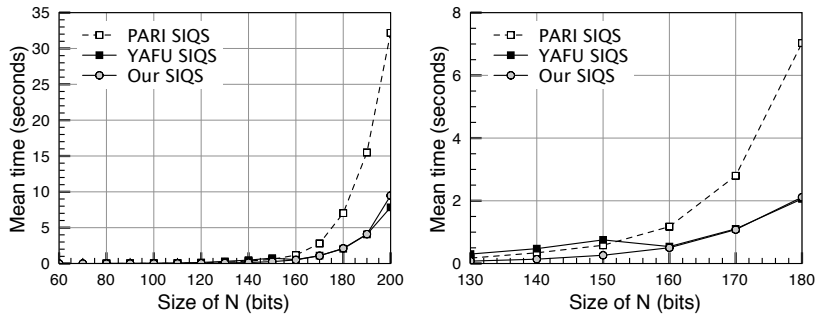

Fig. 9: Comparison of SIQS with two other implementations : PARI's and YAFU's (particularly fast on Opteron). The dip in YAFU's curve at 160 bits is due to the switch from QS/MPQS to SIQS.

3.7 Overall comparison

Fig. 10 gives an overall comparison of the implemented algorithms in our range of interest. SQUFOF is found to remain competitive up to 60 bits but we should mention that our ECM implementation, unlike SQUFOF's, is generic which obviously has an impact on performances. For larger composites, SIQS is clearly the better choice, besting CFRAC across the whole size range.

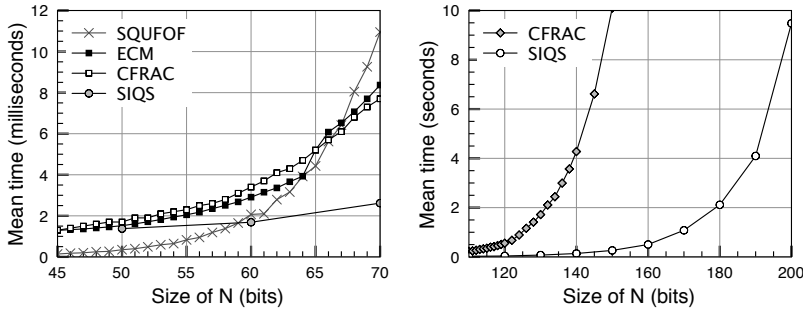

Fig. 10: Comparison of the implemented algorithms over the 45-200 bit range.

4 Concluding remarks

We experimentally assessed the current practicality of factoring methods with an emphasis on small to medium-sized composites with no small factors. From our work, it transpires that :

- SQUFOF is still a viable alternative to ECM, but for tiny composites only for which the "fast-return" variant benefit is then completely negligible.
- McKee's improved Fermat method performs very poorly and is certainly not a viable alternative to SQUFOF.
- While the use of smoothness batches does speed-up CFRAC, it remains not viable compared to sieving methods, even for smaller numbers.
- SIQS can indeed be finely tuned to factor smaller composites (say from 60 or 70 bits to 150 bits). Using QS or MPQS to factor these smaller numbers results in poorer performances.

Finally, our implementations are part of a larger software package available at http://www.lix.polytechnique.fr/Labo/Jerome.Milan/tifa/tifa.html.

References

1. B. Carrier and S. S. Wagstaff, Jr. Implementing the hypercube quadratic sieve with two large primes. In *International Conference on Number Theory for Secure Communications*, pages 51–64, december 2003.

2. D. J. Bernstein. How to find smooth parts of integers, 2004. http://cr.yp.to/papers/sf.pdf.
3. J.F. Biasse. Improvements in the computation of ideal class groups of imaginary quadratic number fields. Submitted to Advances in Mathematics of Computation, 2009.
4. D. Boneh. Twenty years of attacks on the rsa cryptosystem. *Notices of the AMS*, 46:203–213, 1999.
5. R. Brent. Some Integer Factorization Algorithms using Elliptic Curves. Report CMA-R32-85, The Australian National University, 1985.
6. J. Buchmann. A subexponential algorithm for the determination of class groups and regulators of algebraic number fields. In *Séminaire de Théorie des Nombres, Paris 1988–1989*, Progress in Mathematics, pages 27–41. Birkhäuser, 1990.
7. S. Contini. Factoring integers with the self-initializing quadratic sieve, 1997. http://citeseer.ist.psu.edu/contini97factoring.html.
8. T. Kleinjung et al. Factorization of a 768-bit rsa modulus. Cryptology ePrint Archive, Report 2010/006, 2010. http://eprint.iacr.org/2010/006.pdf.
9. J. E. Gower and S. S. Wagstaff Jr. Square form factorization. *Mathematics of Computation*, May 2007.
10. Jr. H. W. Lenstra. Factoring Integers with Elliptic Curves. *Annals of Mathematics*, 126:649–673, January 1987.
11. M. J. Jacobson Jr. Applying sieving to the computation of quadratic class groups. *Math. Comp*, 68:859–867, 1999.
12. K. Gaj et al. Implementing the Elliptic Curve Method of Factoring in Reconfigurable Hardware. In *Cryptographic Hardware and Embedded Systems - CHES 2006*, volume 4249 of *Lecture Notes in Computer Science*, pages 119–133. Springer Berlin, 2006.
13. J. McKee. Speeding Fermat's factoring method. *Mathematics of Computation*, 68(228):1729–1737, October 1999.
14. P. Montgomery. Speeding the Pollard and Elliptic Curve Methods of Factorization. *Mathematics of Computation*, 48:243–264, 1987.
15. M. A. Morrison and J. Brillhart. A method of factoring and the factorization of F_7. *Mathematics of Computation*, 29(129):183–205, January 1975.
16. A. M. Odlyzko. Discrete logarithms in finite fields and their cryptographic significance. In *Advances in Cryptology*, volume 209 of *Lecture Notes in Computer Science*, pages 224–314, 1985.
17. D. Parkinson and M. C. Wunderlich. A compact algorithm for Gaussian elimination over GF(2) implemented on highly parallel computers. *Parallel Computing*, 1:65–73, 1984.
18. C. Pomerance. Analysis and comparison of some integer factoring algorithms. In H. W. Lenstra, Jr. and R. Tijdeman, editors, *Computational Methods in Number Theory, Part I*, pages 89–139. Mathematisch Centrum, Amsterdam, 1982.
19. D. Shanks. Unpublished notes on SQUFOF, circa 1975 (?). http://www.usna.edu/Users/math/wdj/mcmath/shanks_squfof.pdf.
20. R. D. Silverman. The Multiple Polynomial Quadratic Sieve. *Mathematics of Computation*, 48(177):329–339, January 1987.
21. H. C. Williams and M. C. Wunderlich. On the Parallel Generation of the Residues for the Continued Fraction Factoring Algorithm. *Mathematics of Computation*, 48(177):405–423, January 1987.